Test Generation for Intelligent Networks Using Model Checking

André Engels[1], Loe Feijs[1,2], Sjouke Mauw[1]

[1] Eindhoven University of Technology
P.O. Box 513, 5600 MB Eindhoven, the Netherlands
[2] Philips Research Laboratories Eindhoven
Prof. Holstlaan 4, 5656 AA Eindhoven, the Netherlands

{engels,feijs,sjouke}@win.tue.nl

Abstract. We study the use of model checking techniques for the generation of test sequences. Given a formal model of the system to be tested, one can formulate test purposes. A model checker then derives test sequences that fulfill these test purposes. The method is demonstrated by applying it to a specification of an Intelligent Network with two features.

1 Introduction

We will discuss testing, and more specifically the testing of telecommunication systems. Testing is necessary in several phases of a development process. In the first place there is the testing during the specification phase. Central questions in this respect are whether the specification follows the requirements, and whether any logical errors are present in the specification. In telecommunication systems a probable cause for those logical errors is feature interaction [22], that is, the effect that different features (variations on the basic protocol) have on one another. This could for example happen if one feature changes a variable another feature uses or changes as well.

Secondly, it is also necessary to test whether the implementation conforms to the specification. Feature interaction is again an important subject, for example through the sharing of (necessarily finite) resources. It is this second testing phase that will be investigated in this paper

One problem in testing is the creation of a suitable test-set, a set of test traces to be checked. Manual generation of test traces is a lot of work, so it is natural to look for computer support. In relatively small cases this is perfectly feasible: there are techniques and tools that, given a formal specification, generate a complete set of test traces. See for example [24]. Assuming that the implementation has as many states as the specification, a positive result of the test can be considered a correctness proof. In many practical cases there is a so-called state space explosion, that causes the number and/or the length of the traces to be (much) larger than can be dealt with. In this case one has to choose which traces are and which are not to be tested. This selection of interesting traces requires much insight in the problem at hand, so cannot be automated. Still, support in this process will be useful.

Our aim is not to create yet more new tools, but to find and link existing tools that suit this purpose. At present this also means we do not go beyond prototyping. In particular we will try to use tools from Model Checking to generate traces. Model Checking provides us with serious tools with a good theoretical foundation and the possibility to work with large examples (so we can more easily cope with the problems of the state space explosion). Another important advantage is that tools to translate the output of SPIN (the Modelchecker we used) [13] into useful formats, either exist or are easily created. In short, this article will be about the usage of existing tools for the purpose of trace generation for implementation checking of systems. We will use this for systems that are far beyond full state space checking, that is, where a full state space check can not be reached or even approximated.

As an example to apply our methods to we have chosen Intelligent Networks [15]. This is an important application in which conformance testing, as well as other tests, are a necessity. Because of the regular addition of features these tests also have to be repeated during the lifespan of the network. Moreover, these features can have unexpected interactions, so every time a test has to be done of the system as a whole. The addition of features often causes an exponential growth of the state space, so a state space explosion will be almost a certainty. We used a model of a telephone service, with two features: Originating Call Screening (OCS) and Hotline. We have successfully applied our method, and the developed prototype, to this simplified example.

The structure of this paper is as follows. In Sect. 2 we discuss the context of our work by relating it to other approaches. The proposed methodology is explained in Sect. 3. In Sect. 4 we discuss the case study on Intelligent Networks.

2 Related Work

One important approach to testing is *finite state machine exploration*. This is particularly useful for testing the conformance of (local) protocol implementations, where the finite state machine (FSM) models are already frequently used as a specification technique, and where the state space of the implementations is reasonably small. Several strategies to generate efficient test sequences which guarantee a complete test coverage (under certain fault assumptions) are known, such as UIO (unique input-output) sequences, see e.g. [7], [25], [29] and [31]. Certain tool sets (such as the Dutch PTT's Conformance Kit) allow for test-guidance, that is, a labeling mechanism to express that certain transitions deserve special attention (this is a step towards explicit 'test-purposes'), see e.g. [18] for an application. The advantage of finite state machine exploration is the complete coverage obtained (if feasible). This means that the tester does not need to formulate test-purposes and does not have to worry about assigning priorities to test-purposes. But for larger systems such as entire service layers, the approach becomes more and more difficult and the tests will have to be based on explicit test-purposes (as prescribed in the ISO 9646 framework, see [17]).

In [12], test purposes (for conformance testing) are expressed in an informal style, e.g. "send initiate-request to IUT and await abort-request from IUT", or "check the correct execution of process (data transfer)". These test purposes are analyzed by means of a knowledge-based approach and then test-specifications and verdicts are automatically derived from a specification, which is assumed to be given as a finite behavior tree. In [10] test purposes (for service testing) are expressed as walks through a finite state-machine. The state machine is a very abstract model of one of the processes involved (typically the B-party in an asymmetric session protocol). These test purposes are simulated in an interactive way, using a formal specification in PSF and the resulting trace is converted into a test in TTCN (Tree and Tabular Combined Notation) [26] in a partially automated way.

In the TOPIC V2 test approach of [23], also described in [9], test purposes for conformance testing are expressed as Message Sequence Charts (MSCs) [16]. These test purposes are converted into an observer automaton, which is co-simulated with the SDL protocol [6] model, which leads to a constrained graph from which the TTCN test tree is eventually derived in an automatic way.

A very similar approach is the TGV approach of [11], also described in [9], where the behavioral part of a test purpose is expressed as an automaton. The automaton is a directed acyclic graph with a set of distinguished accepting states, but in practice it may be derived from an MSC itself. From the test automaton and a given SDL model a test-graph is generated, using a technique called on-the-fly verification (which means that it is not necessary to have the entire graph in memory). The resulting test case in TTCN is obtained by adding timer management and by taking the asynchronous nature of some of the points of control and observation (PCOs) into account.

In [30] an intelligent network (IN) service creation environment is described where on-line verification of properties is built-in to the environment. Constraints and consistency conditions of the intended service are checked via model checking. The main difference with our own work is the fact that [30] embeds the verification into the design process, essentially working in a white-box fashion, whereas we address the testing problem as a black-box problem (but of course it is uneconomic to build a complete model of the network from scratch, so in practice our approach will have to re-use models taken from the service creation process as well).

It is clear that MSCs play a key role in almost all approaches to protocol testing, either as test-purposes, or otherwise as intermediate representations of test traces. Another test-philosophy was proposed at TACAS '96 by Holzmann and his co-workers: in [14], [2] tools for analyzing MSCs are presented which are primarily considered as 'early fault detection' tools. Sets of MSCs are tested for internal consistency themselves, which means that a testing takes place even before the state-machine models or implementations have been built.

3 Methodology

In this section we will give an overview of our methodology (see Fig. 1). The starting point will be the specification of the system under study. We have taken an SDL ([6], [28]) specification as our input. The first step is to translate this specification (manually) into a form understood by the model checker. Since we used the model checker SPIN [13], we translated this SDL-specification into Promela, the modeling language of SPIN. The structure of a Promela model (several parallel processes, which can communicate both through shared variables and through channels), also fits neatly with SDL-descriptions. The Promela-code was created from the SDL-specification by hand, but a few macros were used to bring the Promela code closer to the SDL-code.

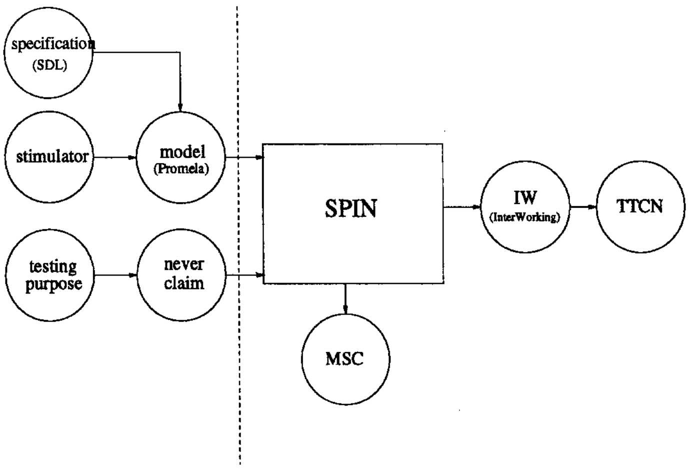

Fig. 1. general method

During this translation, and even during the creation of the specification itself, it is a good thing to already start looking at the testing goals. Sometimes auxiliary variables are necessary to count the number of times a certain step in the process has been taken or is being taken (as the value of such a variable might be part of the testing goal). Also the degree of simplification might differ, depending on what is to be tested.

In this model we also incorporate a so-called stimulation process. This is an added process, that regulates the external inputs and/or the independent actions of the system to be tested. It sends messages to the other processes, either through a specialized channel or through the change of some variable, that normally work as a trigger for performing some activity. For example, in our test case, a model of a telephony service, the stimulation process regulates which calls are to be attempted.

Next, we have to develop a test purpose. This consists of the desired characteristics of the test traces to be developed. Of course, these characteristics have to be chosen such, that the test traces to be found have a high chance of catching implementation errors, that is, they should describe a situation where the system is likely to behave differently from the specification in the case of errors. Because of this, it is important to guess which kind of errors are most likely to occur, or most important to be found.

In their most simple form these testing purposes consist of a property or a set of properties for the final state of the trace, but they could also be more complicated, for example a series of states (distinguished by their properties) that have to be traversed, or an added restriction on the states before the final one. The only consideration is that it must be possible to write the testing purposes down in (temporal) logic.

These two additions (stimulation process and testing purpose) give two ways of controlling the test trace developed. The stimulation process describes the search space for traces, while the testing purpose regulates which kind of traces are actually generated. Of course these two will be connected: On the one hand one can cause the stimulation process to make only those actions happen which bring the testing purpose closer, thus making the number of possibilities checked smaller, or one can disallow the most trivial ways of reaching the testing purpose, thus finding other, possibly more interesting, traces. Finally, one can re-use the same testing purpose by using it together with different stimulators.

We now take a model checker (in our example SPIN), and take the negation of the testing purpose as a so-called never-claim. In normal usage of model checkers the never-claim is an asserted logical invariant of the model, and should therefore never become false (hence the name never-claim). The model checker then runs the model, checking whether the never-claim ever becomes false, and presenting a trace that makes the never-claim false if this is the case. Here we take the negation of the testing purpose, which will cause the trace found to be one in which the negation of the testing purpose is false, and hence the testing purpose itself is reached. In general there will be more than one test trace possible that fulfills the testing purpose. In that case the model checker will make an essentially nondeterministic choice (although some model checkers might allow one to find all traces, or one particular (such as the shortest)).

From an output of SPIN (which contains all the information about the trace that is found) we create Interworkings (IW) [21], a (TUE and Philips) local variant of synchronous MSC-like diagrams. The reason we do this, is that a tool is available [10] for translating such an IW into TTCN, which of course is our final goal. The SPIN-output also contains some MSCs, which can be used for a quick scan of the trace found, and thus can give help for human control of the test generation process.

A scheme of the method can be found in Fig. 1. The part left of the dashed line is mainly manual work, the part on the right is done by tools.

4 Case Study: Testing Intelligent Networks

In this section we use our method in an example from the field of Intelligent Networks (IN). We have modeled a telephony service with two features, OCS (Originating Call Screening) and HOT (Hotline). Using our methods, we will derive a single test trace.

4.1 Intelligent Networks

Because of the ever-growing amount of possibilities of telephone services, a new paradigm for telephony and connected telecommunication has been developed: Intelligent Networks. The following citation from [30] characterizes the IN-concept:

> Intelligent Network (IN) services are customized telephone services, like e.g., 1) 'Free-Phone', where the receiver of the call can be billed if some conditions are met, 2) 'Universal Private Telephone', enabling groups of customers to define their own private net within the public net, or 3) 'Partner Lines', where a number of menus leads to the satisfaction of all desires. The realization of these services is quite complex and error prone.
>
> The current trend in advanced IN services clearly evolves towards decoupling Service Processing Systems from the switch network (see e.g. [8]). The reasons

for this tendency lie in the growing need for decentralization of the service processing, in the demand for quick customization of the offered services, and in the requirement of rapid availability of the modified or reconfigured services. Service Creation Environments for the creation of IN-services are usually based on classical 'Clipboard-Architecture' environments, where services are graphically constructed, compiled, and successively tested. Two extreme approaches characterize the state of the art: The first approach guarantees consistency, but the creation process is strongly limited in its flexibility to compose Service Independent Building Blocks (SIBs) to new services. The second approach allows flexible compositions of services, but there is little or no feedback on the correctness of the service under creation during the development: the validation is almost entirely located after the design is completed. Thus the resulting test phase is lengthy and costly.

For more information on IN we refer to e.g. [1], [4], [8] or [15].

In [32] the need for service testing in the context of IN is explained and a framework for testing telecommunication services is presented, where it is stressed that next to the service itself, the underlying platform and the already existing services should be tested as well.

4.2 A Simple Model

We will model an Intelligent Network with two special services (which we call 'features'), namely OCS (Originating Call Screening) and a simplified version of HOT (Hotline). In Originating Call Screening phone calls can be blocked by the receiver depending on the originator of the call. In Hotline the dialing of often used numbers is made easier by causing another (smaller) number to result in the same connection. In our model the Hotline will be established on dialing any number that is not a service number. Adding a feature can be done without much problems (although it will increase the size of the space state, and thus might cause the generation time for the test trace to increase slightly).

In our model we will decouple the SSP (Service Switching Point), which connects the user with the services, and the SCP (Service Control Point), which physically contains the services. This decoupling is suggested within the literature (see above), because the implementation and maintenance of the system is easier if the (stable) basic functions are decoupled from the (dynamically added) features. For the model there would be only little change if we had not implemented this decoupling, but for the test trace generation this might very well be important, since an overload of the connection between SSP and SCP might be a cause of errors.

In Fig. 2 we see the architecture of an Intelligent Network as it is represented by our model. The SSP is responsible for the connections between the telephones, and the connections between the telephones and the SCP. It can be modeled in for example SDL.

Through a special channel, ss7, the SSP is connected to the SCP. The SCP checks which, if any, features have to be used in a given call. In our model the SDP, which does the maintenance of the features, that is, keeping track of which features are enabled for whom and how they are configured, and the SCP have been combined into one process. The SCP is normally modeled by Service Independent Building Blocks (SIBs) [30].

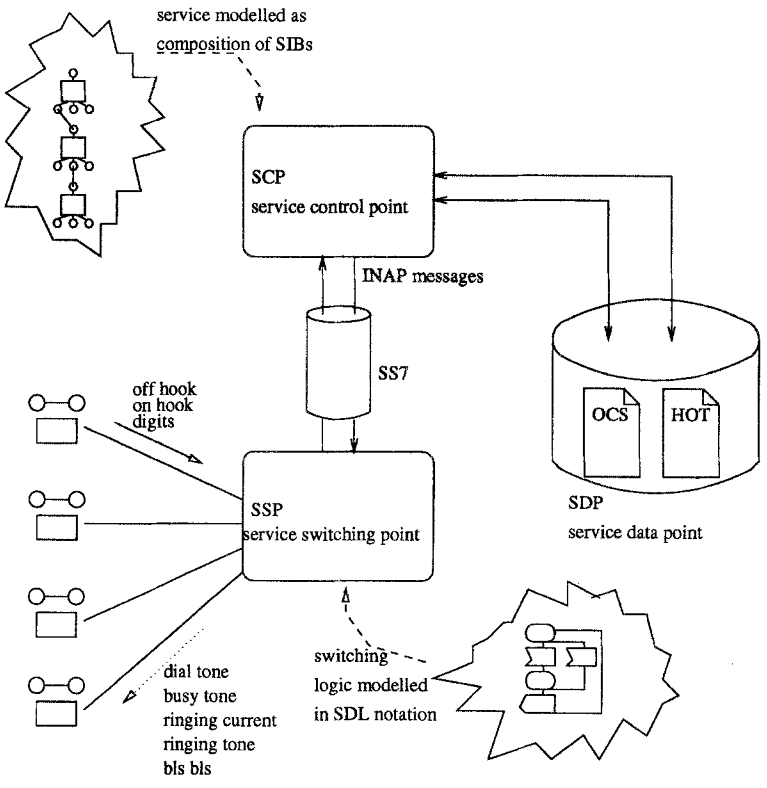

Fig. 2. IN architecture

In Fig. 3 we see an SDL-model of the SSP as we have modeled it. This model is based on [27]. Identifier A here stands for the person attempting to make the call, B for the person being called. The SSP is idle until there is an off-hook message (A off hook), after which a dial tone is sent to phone A, and then the SSP is in the await_digits-state. This one can end in two ways, namely by A putting the phone on hook, and by A dialing a number. In the first case the (attempted) call ends, and the SSP becomes idle again. In the second case, the SSP checks whether the line called is busy, if it is it generates a busy tone and waits for an on_hook, otherwise the second phone starts ringing and a conversation is attempted. The rest of the figure reads likewise.

This scheme is simplified from the form we used in our model in a few ways: Firstly, there can be more than one attempt for making a talk. Because of this, many copies of this scheme are running at the same time, one for each call attempted. Secondly, not all actions (the sending of tones and talk across the telephone lines) are shown. In the third place, this only specifies the behavior in absence of any special features. The presence of features influences the effect in the following ways:

- Hotline changes the line with which a connection is attempted
- OCS makes 'called line busy?' true even when the line is not busy
- After the digits are dialed, it is first checked whether they are the digits for adding

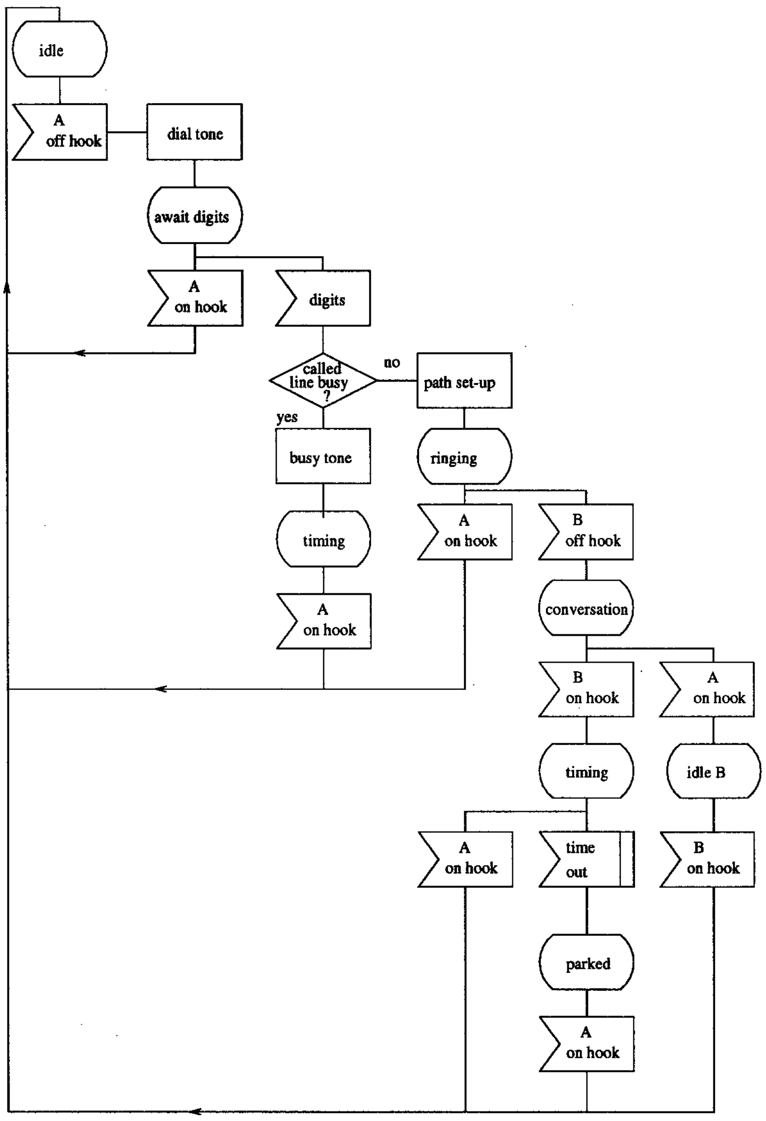

Fig. 3. SDL-model of the SSP

or removing a feature. If so, the feature is enabled/disabled/changed, and the phone waits for an on_hook

We translated this SDL-model into PROMELA-code. We will not be giving the complete (5-page) PROMELA-code of our model here, but only a few representative parts. First we give a part of the SSP-code:

```
STATE(ringing)
:: signal[A]?on_hook  -> atomic {
```

```
                                            line[A] = silent;
                                            line[B] = silent;
                                            busy[A] = false;
                                            busy[B] = false;
                                            NEXTSTATE(idle)
                                          }
     :: intern[B]?off_hook -> atomic {
                                            line[A] = blabla(B);
                                            line[B] = blabla(A);
                                            BCP_account[A]++;
                                            NEXTSTATE(conversation)
                                          }
     ENDSTATE
```

In Fig. 3 this state ('ringing') can be found a bit above the middle (below the action 'path set-up'). Take special notice of the variable BCP_account[A]. It is incremented each time A has made a successful attempt to engage in a talk. It has no function in the model, but we include it in order to be able to have 'A has made n talks' as (part of) our testing goal.

The model of the SCP is shown below (OCS is a 5 times 5 boolean array, HOT is an integer array of length 5):

```
do
:: ss7?feature(A,B) -> if
                       :: B/100 == 66 -> HOT[A] = (B - 6600)
                       :: B/100 == 88 -> OCS[B - 8800,A] = true
                       :: B/100 == 89 -> OCS[B - 8900,A] = false
                       fi
:: ss7?check(A,B) ->   s = OCS[A,B];
                       ss7!checked(!s,_);
                       if
                       :: (s == 1) -> IN_account[B]++
                       :: else -> skip
                       fi
:: ss7?lookup(A,B) ->  if
                       :: (HOT[A] == A) -> ss7!lookuped(A,B)
                       :: (HOT[A] != A) -> ss7!lookuped(A,HOT[A])
                       fi
od
```

First, there are a few arrays: OCS[A,B] is true iff B is blocking messages from A, HOT[A] is the Hotline A has (if it has A as its value, A does not have a Hotline). These arrays are filled in the snipped part.

The SCP gets its orders from the SSP through the ss7-channel. The message 'feature(A,B)' adds or removes a feature, the message 'check(A,B)' asks whether A is allowed to call B, and the message 'lookup(A,B)' tells A has dialed the number B, and asks with which phone a connection should actually be attempted.

Through this same channel the SCP sends the results back to the SSP. If the order was a check, it sends 'checked(!s,_)', where s is true iff B is blocking messages from A, while the second one ('_') is a dummy variable, which is only needed because the ss7 is a 3-variable channel. If the order was a lookup, then 'lookuped(A,H)' is sent, where H is A's hotline if any, and B otherwise, so in fact it is sending the number that will be the real receiver of the message.

As before, there is an auxiliary variable: IN_ACCOUNT[B], which counts the number of calls to B that have been blocked by B.

The stimulation process controls the amount of non-determinism in the system. An example of a stimulation process can be found below:

```
proctype stimula()
{
        call[2]!6603;           /* 2 has Hotline to 3   */
        call[4]!8801;           /* 1 should not call 4 */
        do
        :: call[4]!8901         /* 1 may call 4 again   */
        :: call[1]!4
        od
}
```

The action call[A]!B sends a message to phone A, telling it to attempt to make a call in which it dials number B. So the stimulation process above first orders phone 2 to create a Hotline to 3, then orders phone 4 to create an OCS towards 1, and then goes through a cycle, every time either ordering phone 4 to stop its OCS towards 1, or ordering phone 1 to attempt a call to phone 4.

This is of course just one example of a stimulation process. We have worked with several different processes in order to get different traces.

4.3 Generating a Test Sequence

As an example, we will generate an interesting trace. As a working hypothesis we assumed that problems were likely to arise due to mistaken allocation of shared resources, especially if some resource was used too extensively. This leads to testing goals like 'There are n SDP-accesses taking place' However, because our main goal was the testing of the feasibility of the general method, we have only used the simplest cases in practice, such as:

- Phone A has made a successful call
- Phone A has made two successful calls
- Phone A and B have been connected in a successful call
- An SDP-access is taking place

In practice more complex situations have to be checked. This might cause a longer computation time, because the minimal length of a trace that has the desired properties is longer, and the testing purpose is more complicated. Neither seems to be really problematic, though.

As an example we take the testing goal 'Phone 1 has made a successful call', with the stimulation process as described above.

In SPIN the testing goal can be implemented as a "never-claim".

```
#define ALWAYS(P) never { do :: P :: !(P) -> break od }
#define CLAIM(A) (BCP_account[A] < 1)
ALWAYS(INV(1))
```

SPIN will look for traces in which the process defined by the never-claim has ended. In our definition of ALWAYS(P) this means that P has been false at some place of the trace - in fact, at the last step of the trace. So if we make our claim ALWAYS(P), then SPIN will be looking for a trace that ends with a situation in which P is NOT true. As we want to have a trace in which phone 1 has made a (successful) call, this P must be 'Phone 1 has not made a call', which, because of the addition of the variable BCP_account[A] into our model, simply translates into 'BCP_account[1]<1'.

This model was run using SPIN (XSpin). It did find a trace to a state in which the BCP-account of telephone 1 is at least 1. The main part of the SPIN-output consists of listings of the following form. It is in fact a complete list of the actions taken by the various processes, with some information added (the line of code where the action is described, the value of variables that have changed, etcetera).

```
1: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
2: proc 1 (:init:) line 323 "pan_in" (state 1) [(run phone(0))]
3: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
4: proc 2 (phone) line  93 "pan_in" (state 1) [self = self]
          phone(2):friend = 0
          phone(2):state = 0
          phone(2):self = 0
5: proc - (:never:) line 319 "pan_in" (state 1) [((BCP_account[1]<1))]
6: proc 2 (phone) line  94 "pan_in" (state 2) [((self==0))]
etc.
```

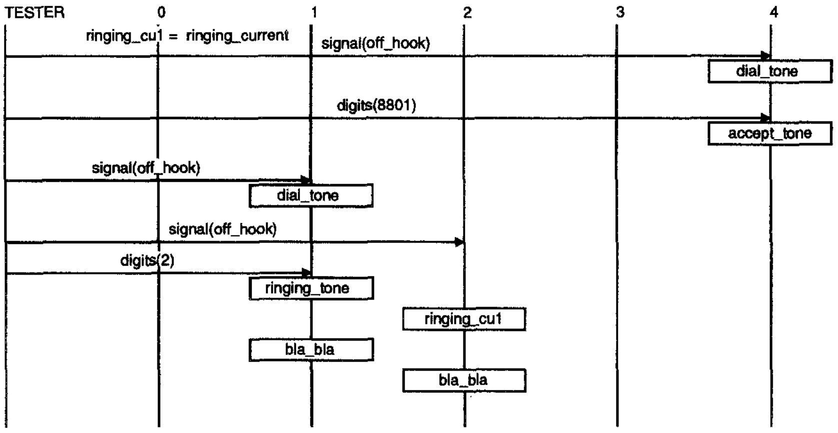

Fig. 4. Interworking of test run after inversion

XSpin enables us to inspect this trace as an MSC, which we will not display. We used a series of Unix shellscripts and existing tools to transform the trace into an Interworking. To this Interworking we applied an 'inversion' [10], which transforms the output lines of the various processes into input lines. This facilitates the use in a testing environment, because we can now regard them as orders to do certain actions, instead of the actions themselves. We decided to receive line-states as (observation) actions. This resulted in the Interworking shown in Fig. 4.

Tools exist to translate this into TTCN. For the case at hand the TTCN looks as follows:

```
+-------------------------------------------------------------------------+
|Test Case FEATURE_INTERACTION_TEST 1                                     |
+-------------------------------------------------------------------------+
|Test Case Name : FIT 1                                                   |
|Group          : \1                                                      |
|Purpose        : 1st demo use SPIN FI TESTING                            |
|Default        :                                                         |
|Comments       :                                                         |
+-------------------------------------------------------------------------+
|Nr | Label | Behavior Descriptions       | Constraints Ref | Verdict     |

              4!signal(off_hook)
               [line 4 = dial_tone]                         (PASS)
                4!digits(8801)
                 [line 4 = accept_tone]                     (PASS)
                  1!signal(off_hook)
                   [line 1 = dial_tone]                     (PASS)
                    2!signal(off_hook)
                     1!digits(2)
                      [line 1 = ringing_tone]               (PASS)
                       [line 2 = ringing_current]           (PASS)
                        [line 1 = bla_bla]                  (PASS)
                         [line 2 = bla_bla]                 PASS
                         [OTHERWISE]                        FAIL
                        [OTHERWISE]                         (FAIL)
                       [OTHERWISE]                          (FAIL)
                      [OTHERWISE]                           (FAIL)
                   [OTHERWISE]                              (FAIL)
                 [OTHERWISE]                                (FAIL)
               [OTHERWISE]                                  (FAIL)      |
+-------------------------------------------------------------------------+
```

In general it is not the case that the TTCN generated from the trace in such a straightforward manner is directly correct as a test. The problem is the correct assignments of verdicts to the alternatives, all of which are made FAIL initially. We discuss three approaches to deal with this problem.

The first approach is as follows: subdivide the trace into two parts, an initial part which serves for setting-up the services and contextual connections, followed by a second part, usually much shorter, which characterizes the intended behaviour of the system. For example in the test case FIT1 given above, the two observation actions [line1 = bla_bla] and [line2 = bla_bla] should be interpreted as a characterisation of the intended behaviour (call established), so the alternatives of the second part could keep the assigned FAIL verdicts. The verdicts of the alternatives of the steps of the first part can be turned into INCONCLUSIVE.

The second approach is a further refinement of this. The generated TTCN is only considered as a draft of the correct test case, which is to be obtained by checking the verdicts and adding more alternatives (some of which may also get PASS). This is the approach of [10], where it is shown in detail how a simulator is used (during multiple runs of the simulator) to find out in how far the crucial steps are deterministic, and if not, what the interesting alternative behaviours are (in the step-by-step method of [10], these are the steps 13, simulate alternatives and 14, complete the TTCN description).

The third approach to the problem of verdict assignment to alternatives is to adapt the Modelchecker and make it produce trees or graphs rather than sequences. This is the approach of the tool TGV [11].

So far we have only used temporal claims of a particularly simple kind, viz. invariants (such as ALWAYS(INV(1))). So we ought to discuss whether temporal claims in general are useful as well. In the classical usage of a Modelchecker, i.e. for verification purposes, it will attempt to falsify claims like: "*it is always true that when the sender transmits a message, the receiver will eventually accept it*". For test generation purposes however, the temporal claim could be used to say for example: "*it is always true that when A is in state trying-to-reach-B, the SSP (Service Switching Point) will eventually connect A and B*". Of course this claim need not be true, e.g. because it is precisely the purpose of certain services (like OCS) to prevent connections from A to B to happen. Therefore, such a temporal claim, when falsified, results a trace leading to a state where this 'prevention' service has been put into operation.

5 Conclusions

Model checking can be useful as a technique for generating test traces. This can be done using existing tools, at least on prototype level. A reservation has to be made on the point of the scaling-up of the tools, because we have only tested small examples. Also the time and memory consumption of the method have not yet been investigated. If we want to use this in practice we will probably need more specialized tools, and we must be able to connect them to service-creation environments.

We found that our way of working (selecting a trace leading to an interesting state) is a promising one. This way, a part of the hard work of the creation of test traces can be automated. Traces can be selected to agree with given testing purposes without having to step down too far in abstraction.

A restriction to the applicability of the method, at least in the current form, is that the application to be tested should react deterministically to the test input. The reason for this is that otherwise a trace in which an error has occured cannot be distinguished from one in which the internal non-determinism has caused the system to react different from the derived test trace, but still within the specification. If the system is not deterministic, the method is still useful, but in that case more manual work is needed to complete the test case.

Our method supports a part of the test traject, namely the derivation of a test trace from a given test purpose. Formulating the test purpose, the stimulation process and the model remains a task that has to be done by hand, and requires an amount of domain specific knowledge.

Another way of working might be introducing deliberate errors in the SPIN program, which are supposed to model possible errors in the design, and creating a trace in which the error occurs. We could call this 'negative testing', because in this case we are constructing traces we want the real design NOT to be able to follow, while in the constructs given until now, we wanted the design to follow the trace we gave it. An example of such a negative test is a trace that leads to a connection between two subscribers while the called party is refusing calls from the calling party by using the OCS feature.

However, we think that positive testing is more suitable to combination with the given method, because for negative testing we need to make many more assumptions about the kind of errors that might occur. In negative testing we need a rather specific idea about WHAT errors can occur, in positive testing we only need to hypothesize on WHEN they occur. When looking for specific errors, negative testing is the way to go, but if the purpose is to make a general check of a system, positive testing is much more useful.

One objection to our method could be that in order to generate a trace satisfying the property checked, the Modelchecker risks searching the entire state space, which may be infeasible (the problem of the state space is often stated as an argument for the need of testing in the first place). Although this is true in principle, the important observation is that a Modelchecker such as SPIN has powerful techniques built into it (such as the supertrace algorithm) to cope with the state space problem. In our opinion it is important that (if testing cannot be made superfluous by other means, for any reason whatsoever), the testers should use powerful and high-level tools as well; in particular this holds for the intermediate situation where fully automated testing is infeasible and where fully manual test generation is too costly.

Acknowledgments: The authors wish to thank Rob Gerth, Bart Knaack, Thierry Jeron and the anonymous reviewers for the discussions and the help that contributed to the work presented in this paper.

References

1. J. Aitken. Intelligent networks, seminar, logica uk ltd., london, april 26–27, 1995, 1995.
2. R. Alur, G. Holzmann, and D. Peled. An analyzer for message sequence charts. In Margaria and Steffen [20], pages 35–48.
3. B. Baumgarten, H. Burkhardt, and A. Giessler, editors. *Testing of communicating systems*. Chapman & Hall, 1996.
4. P. Bohacek and J.N.White. Service creation: The real key to intelligent network revenue. In *Proc. Workshop Intelligent Networks '94*, Heidelberg, May24–26 1994.
5. A. Cavali and S. Budkowski, editors. *Proceedings of IWPTS '95*, 9 rue Charles Fourier 91011 Evry Cedex, 1995. Institut National Télécommunications.
6. CCITT. Recommendation z100, specification and description language (sdl). Recommendation AP IX-35-E, International Telegraph and Telephone Consultative Committee, Geneva, 1988.
7. W. Chun. Improvements on uio sequence generation and partial uio sequences. In Linn and Üyar [19].
8. E. Crabill and J. Kukla. Service processing systems for at&t's intelligent network. *AT&T Techn. Journal*, 73(6):39–47, 1994.
9. L. Doldi, V. Encontre, J. Fernandez, T. Jéron, S. L. Bricquir, N. Texier, and M. Phalippou. Assessment of automatic generation methods of conformance test suites in an industrial context. In Baumgarten et al. [3], pages 347–361.
10. L. Feijs and M. Jumelet. A rigorous and practical approach to service testing. In Baumgarten et al. [3], pages 175–190.

11. J. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on the fly verification techniques for the generation of test suites. In A. Alur and I. Henzinger, editors, *Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA*, number 1102 in Lecturen Notes on Computer Science. Springer, July 1996.
12. A. Guerrouat and H. König. Automation of test case derivation in respect to test purposes. In Baumgarten et al. [3], pages 207–222.
13. G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall/AT&T, 1991.
14. G. Holzmann. Early fault detection tools. In Margaria and Steffen [20], pages 1–13.
15. ITU. *Principles of Intelligent Network Architecture, CS1 Recommendations*. ITU, 1993.
16. ITU-TS, Geneva. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, 1994.
17. K. Knightson. *OSI protocol conformance testing*. McGraw-Hill, 1993.
18. E. Kwast. An exercise in test generation for telecommunication services. In J. Springintveld, editor, *Second Dutch Testing Conference (handout)*, volume 03-95-5003A. Katholieke Universiteit Nijmegen, 1996.
19. R. Linn, Jr. and M. U. Üyar, editors. *Protocol specification, testing and verification XII*. Elsevier Science Publishers (North-Holland), 1992.
20. T. Margaria and B. Steffen, editors. *Tools and Algorithms for the Construction and Analysis of Systems*, number 1055 in Lecturen Notes on Computer Science. Springer Verlag, 1996.
21. S. Mauw, M. van Wijk, and T. Winter. A formal semantics of synchronous Interworkings. In O. Færgemand and A. Sarma, editors, *Proceedings of the Sixth SDL Forum, Darmstadt*. Elsevier Science Publishers (North-Holland), 1993.
22. C. Middelburg. A simple language for expressing properties of telecommunication services and features. publication 94-PU-356, PTT Research, 1994.
23. J. Montiel, R. Roth, and A. Donaldson. Methods for QoS verification and protocol conformance testing in IBC results and further recommendations. topic R2088 TOPIC, Deliverable 15, DocR2088/DAT/TMS/DS/P/015/b1, RACE project, 1994.
24. R. Nahm. Conformance testing based on formal description techniques and message sequence charts. Technical report, Institut für Informatik, Universität Bern, 1995.
25. K. Naik. Fault-tolerant UIO sequences in finite state machines. In Cavali and Budkowski [5], pages 207–220.
26. *OSI. Conformance testing methodology and framework, Part 3: The Tree and Tabular Combined Notation (TTCN). ISO/IEC DIS 9646-3*, 1990.
27. F. Redmill and A. Valdar. *SPC digital telephone exchanges, revised edition*. Number 21 in IEE Telecommunication series. Peter Peregrinus Ltd., 1994.
28. R. Saracco, J. Smith, and R. Reed. *Telecommunications systems engineering using SDL*. Elsevier Science Publishers (North-Holland), 1995.
29. D. Sidhu and T. Leung. Formal methods in protocol testing, a detailed study. *IEEE transaction on Software Engineering*, 15(4):413–426, 1989.
30. B. Steffen, T. Margaria, A. Classen, V. Braun, and M. Reitenspiess. An environement for the creation of intelligent network services. In *Intelligent Networks: IN/AIN Technologies, Operations, Services, and Applications - A Comprehensive Report*, pages 287–300. IEC - International Engineering Consortium, 1996.
31. X. Sun, Y. Shen, F. Lombardi, and D. Sciuto. Protocol conformance testing by discriminating uio sequences. In B. Johnsson, J. Parrow, and B. Pehrson, editors, *Protocol specification, testing and verification XI*. Elsevier Science Publishers (North-Holland), 1991.
32. G. Vermeer, M. Witteman, and J. Kroon. A framework for testing telecommunication services. In Cavali and Budkowski [5], pages 129–140.